# ASSIGNMENT NO.7

**Title: Object detection using Transfer Learning of CNN architectures**

**Aim**: Object detection using Transfer Learning of CNN architectures

Load in a pre-trained CNN model trained on a large dataset

Freeze parameters (weights) in model's lower convolutional layers

Add custom classifier with several layers of trainable parameters to model

Train classifier layers on training data available for task

Fine-tune hyper parameters and unfreeze more layers as needed

**Theory:**

**Transfer learning** is the reuse of a pre-trained model on a new problem. It's currently very popular in deep learning because it can train deep neural networks with comparatively little data. This is very useful in the data science field since most real-world problems typically do not have millions of labeled data points to train such complex models.

Transfer learning, used in machine learning, is the reuse of a pre-trained model on a new problem. In transfer learning, a machine exploits the knowledge gained from a previous task to improve generalization about another. For example, in training a classifier to predict whether an image contains food, you could use the knowledge it gained during training to recognize drinks.

**A pre-trained model** refers to a model or a saved network created by someone else and trained on a large dataset to solve a similar problem. AI teams can use a pre-trained model as a starting point, instead of building a model from scratch. Examples of successful large-scale pre-trained language models are Bidirectional Encoder Representations from Transformers (BERT) and the Generative Pre-trained Transformer (GPT-n) series.

**PyTorch** is an open source machine learning (ML) framework based on the Python programming language and the Torch library. Torch is an open source ML library used for creating deep neural networks and is written in the Lua scripting language. It's one of the preferred platforms for deep learning research.

The two main features of PyTorch are:

- Tensor Computation (similar to NumPy) with strong GPU (Graphical Processing Unit) acceleration support
- Automatic Differentiation for creating and training deep neural networks

In PyTorch, modules are used to represent neural networks.

- Autograd
- Optim
- nn

**Advantages of Transfer learning:**

1. Reduced Training Time: Transfer learning saves time by starting with a pre-trained model.
2. Improved Performance: It enhances model performance by building on existing knowledge.
3. Better Generalization: Pre-trained models provide robust feature representations for new data.
4. Handling Small Datasets: Effective when data is limited, preventing overfitting.
5. Domain Adaptation: Adapts models to different data distributions.
6. Feature Extraction: Extracts useful features from pre-trained models.

Transfer learning has a wide range of applications across various domains. Some notable applications include:

1. Computer Vision:
   - Object recognition and classification.
   - Image segmentation.

2. Natural Language Processing (NLP):
   - Text classification.
   - Sentiment analysis.

3. Speech Recognition:
   - Automatic speech recognition.
   - Speaker identification.

**Caltech-101** consists of pictures of objects belonging to 101 classes, plus one background clutter class. Each image is labelled with a single object. Each class contains roughly 40 to 800 images, totalling around 9k images. Images are of variable sizes, with typical edge lengths of 200-300 pixels. This version contains image-level labels only. The original dataset also contains bounding boxes.

- Additional Documentation: Explore on Papers With Code north_east
- Homepage: https://doi.org/10.22002/D1.20086
- Source code: tfds.datasets.caltech101.Builder
- Versions

3.0.0: New split API (https://tensorflow.org/datasets/splits)
3.0.1: Website URL update
3.0.2 (default) nights_stay: Download URL update

- Download size: 131.05 MiB
- Dataset size: 132.86 MiB • Auto-cached (documentation): Yes
- Splits:

| Split | Examples |
|---|---|
| 'test' | 6,084 |
| 'train' | 3,060 |

The **ImageNet** project is a large visual database designed for use in visual object recognition software research. More than 14 million images have been hand-annotated by the project to indicate what objects are pictured and in at least one million of the images, bounding boxes are also provided.
The ImageNet dataset consists of more than 14M images, divided into approximately 22k different labels/classes. However the ImageNet challenge is conducted on just 1k high-level categories (probably because 22k is just too much).
ImageNet contains more than 20,000 categories, with a typical category, such as "balloon" or "strawberry", consisting of several hundred images. The database of annotations of thirdparty image URLs is freely available directly from ImageNet, though the actual images are not owned by ImageNet.

**Basic Steps for Transfer Learning:**

1.Choose a Pre-trained Model: Select a pre-trained model that is suitable for your target task.

2 Prepare Your Data: Organize and preprocess your dataset in a format compatible with the pre-trained model's requirements.

3 Customize the Model: Modify the top layers of the pre-trained model to match the number of classes in your specific task.

4 Transfer Knowledge: Initialize the model with pre-trained weights and fine-tune it on your dataset.

5 Training and Evaluation: Train the modified model on your dataset and evaluate its performance.

6 Fine-Tuning (Optional): Optionally, you can fine-tune more layers of the model or experiment with hyperparameters to optimize performance.

**Data augmentation** is a technique used to artificially increase the diversity of a dataset by applying various transformations to the existing data. Common transformations include rotation, scaling, cropping, flipping, and color adjustments. This technique is widely used in computer vision and natural language processing to reduce overfitting, enhance model generalization, and improve the model's robustness.
Data augmentation is particularly useful in transfer learning for several reasons. It increases the effective size of the target dataset, making it less likely for the model to overfit when finetuning on a small dataset.

**Preprocessing** is needed in transfer learning to ensure that the input data is in a format compatible with the pre-trained model. This may involve resizing images to match the input size of the pre-trained model, normalizing pixel values, and other necessary transformations. Preprocessing also helps ensure consistency between the source and target datasets, making it easier for the model to learn relevant features.

**PyTorch** Transforms is a module in the PyTorch library that provides a collection of image transformations and data augmentation techniques. It allows users to easily apply various transformations to their data, making it suitable for training deep learning models, including in the context of transfer learning. Transforms can be used to resize, crop, normalize, and apply other data preprocessing steps to ensure data compatibility with the chosen pre-trained model and to augment the data for better model performance

These operations are typically applied to an input image to prepare it for feeding into a neural network. Let's break down each command:

1. **RandomResizedCrop**(size=256, scale=(0.8, 1.0)): This operation first resizes the image to a random size within the specified range (between 80% and 100% of the original size) and then randomly crops it to the specified size (256x256 pixels in this case). Random resizing and cropping introduce variability into the data, which can help the model generalize better.
2. **RandomRotation**(degrees=15): It randomly rotates the image by an angle between 15 and +15 degrees. This introduces robustness to different orientations of objects in the images.
3. **ColorJitter**(): This command performs random color adjustments, such as changes in brightness, contrast, saturation, and hue. This augments the dataset with variations in lighting conditions, making the model more robust.
4. **RandomHorizontalFlip**(): It randomly flips the image horizontally with a 50% chance. This helps the model learn features that are invariant to left-right orientation.
5. **CenterCrop**(size=224): After the random resizing and cropping, this operation performs a centered crop to further resize the image to a fixed size of 224x224 pixels. This size is a common standard for many neural network architectures.
6. **ToTensor():** This command converts the image data into a tensor format, which is a fundamental data type used by deep learning frameworks like PyTorch. It also normalizes the pixel values to the range [0, 1].
7. **Normalize**: Although it's listed without parameters, this likely refers to applying data normalization, which scales the pixel values to have a mean of 0 and a S.D. of 1.

In PyTorch, when training a deep learning model, you often divide your dataset into three parts: the training set, the validation set, and the test set. Data transforms are applied to the training set to augment and preprocess the data. However, when it comes to the validation set, you typically want to apply only data preprocessing, without random augmentations, to ensure a fair evaluation of your model's performance.

**VGG-16** is a deep convolutional neural network model that was introduced by the Visual Geometry Group (VGG) at the University of Oxford. It is known for its simplicity and effectiveness in image classification tasks. VGG-16 is part of the VGG family of models, which includes variations with different depths, such as VGG-19.

In PyTorch, you can use the VGG-16 model from the torchvision library, which provides a collection of pre-trained models for computer vision tasks.

Explanation of the VGG-16 model:

1. **Input Layer:** VGG-16 takes input images of size 224x224 pixels.
2. **Convolutional Layers:** It consists of 13 convolutional layers with 3x3 filters. These layers learn various image features through a series of convolutions and maxpooling operations.
3. **Fully Connected Layers:** After the convolutional layers, there are three fully connected layers. The first two have 4,096 neurons each, and the last one has the same number of output neurons as the number of classes in the classification task (e.g., 1,000 for ImageNet).
4. **Activation Functions:** ReLU (Rectified Linear Unit) activations are used after each convolutional and fully connected layer, except for the output layer.
5. **Pooling Layers:** Max-pooling layers follow several convolutional layers to downsample the spatial dimensions of feature maps.
6. **Softmax Output:** The output layer typically uses the softmax activation function to produce class probabilities.

**Steps/ Algorithm**

Dataset link and libraries : https://data.caltech.edu/records/mzrjq-6wc02 separate the data into training, validation, and testing sets with a 50%, 25%, 25% split and then structured the directories as follows:

/datadir

/train

/class1

/class2

. /valid

/class1

/class2

. /test

/class1
/class2

.

Libraries required : PyTorch torchvision import transforms torchvision import datasets torch.utils.data import DataLoader torchvision import models torch.nn as nn torch import optim

Create Datasets and Loaders : data = {

'train':(Our name given to train data set dir created ) datasets.ImageFolder(root=traindir, transform=image_transforms['train']), 'valid':

datasets.ImageFolder(root=validdir, transform=image_transforms['valid']),

} dataloaders =

{

'train': DataLoader(data['train'], batch_size=batch_size, shuffle=True), 'val': DataLoader(data['valid'], batch_size=batch_size, shuffle=True)

}

Load Pretrain Model : from torchvision import models

model = model.vgg16(pretrained=True) Freez all the

Models Weight

for param in model.parameters(): param.requires_grad = False

Add our own custom classifier with following parameters : Fully connected with ReLU activation, shape = (n_inputs, 256) Dropout with 40% chance of dropping

Fully connected with log softmax output, shape = (256, n_classes) import torch.nn as nn

# Add on classifier model.classifier[6] = nn.Sequential( nn.Linear(n_inputs,

256), nn.ReLU(),

nn.Dropout(0.4), nn.Linear(256, n_classes), nn.LogSoftmax(dim=1)) Only

train the sixth layer of classifier keep remaining layers off . Sequential(

(0): Linear(in_features=25088, out_features=4096, bias=True) (1): ReLU(inplace) :

Dropout(p=0.5)

: Linear(in_features=4096, out_features=4096, bias=True) (4): ReLU(inplace)

: Dropout(p=0.5)

: Sequential(

(0): Linear(in_features=4096, out_features=256, bias=True) (1): ReLU()

: Dropout(p=0.4)

: Linear(in_features=256, out_features=100, bias=True) (4): LogSoftmax())

)

Initialize the loss and optimizer criteration = nn.NLLLoss() optimizer

= optim.Adam(model.parameters())

Train the model using Pytorch for epoch in range(n_epochs): for data, targets in trainloader:

# Generate predictions out = model(data)

# Calculate loss loss =

criterion(out, targets)

# Backpropagation loss.backward()

# Update model parameters optimizer.step()

Perform Early stopping

Draw performance curve Calculate Accuracy pred =

torch.max(ps, dim=1) equals = pred == targets

# Calculate accuracy accuracy

= torch.mean(equals)

**INPUT CODE:**

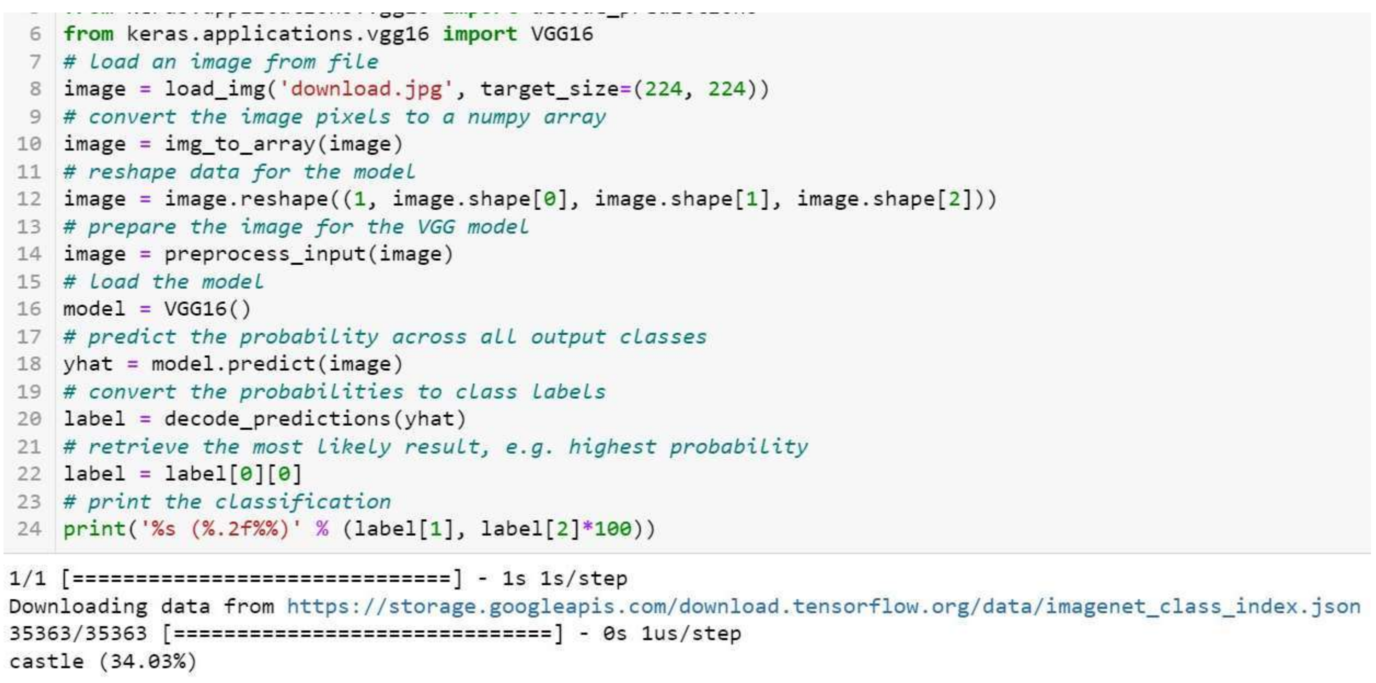
```
# example of using a pre-trained model as a classifier from
tensorflow.keras.preprocessing.image import load_img from
tensorflow.keras.preprocessing.image import img_to_array from
keras.applications.vgg16 import preprocess_input from
keras.applications.vgg16 import decode_predictions from
keras.applications.vgg16 import VGG16
# load an image from file image =
load_img('download.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array image = img_to_array(image) #
reshape data for the model image = image.reshape((1, image.shape[0],
image.shape[1], image.shape[2]))
# prepare the image for the VGG model image
= preprocess_input(image)
# load the model model
= VGG16()
# predict the probability across all output classes yhat
= model.predict(image)
# convert the probabilities to class labels label
= decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0] # print the classification
print('%s (%.2f%%)' % (label[1], label[2]*100))

# load an image from file image =
load_img('download2.png', target_size=(224, 224))
# convert the image pixels to a numpy array image = img_to_array(image) #
reshape data for the model image = image.reshape((1, image.shape[0],
image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)
# load the model model
= VGG16()
```

```
# predict the probability across all output classes yhat
= model.predict(image)
# convert the probabilities to class labels label
= decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0] # print the classification print('%s
(%.2f%%)' % (label[1], label[2]*100))

# load an image from file image =
load_img('download3.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array image = img_to_array(image) #
reshape data for the model image = image.reshape((1, image.shape[0],
image.shape[1], image.shape[2]))
# prepare the image for the VGG model image
= preprocess_input(image)
# load the model model
= VGG16()
# predict the probability across all output classes yhat
= model.predict(image)
# convert the probabilities to class labels label
= decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0] # print the classification print('%s
(%.2f%%)' % (label[1], label[2]*100))
```

**OUTPUT**

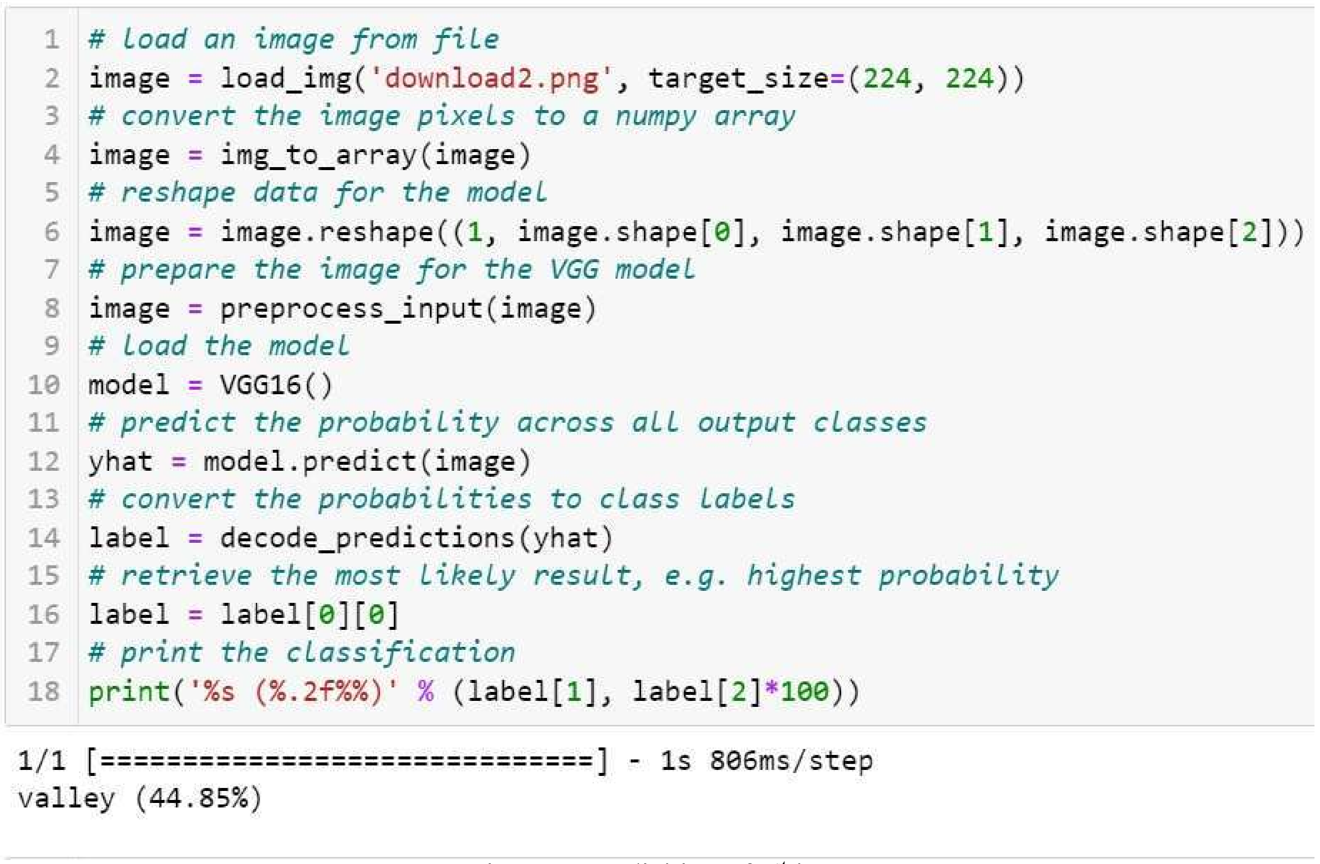


```
from keras.applications.vgg16 import VGG16
# load an image from file
image = load_img('download.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array
image = img_to_array(image)
# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)
# load the model
model = VGG16()
# predict the probability across all output classes
yhat = model.predict(image)
# convert the probabilities to class labels
label = decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0]
# print the classification
print('%s (%.2f%%)' % (label[1], label[2]*100))
```

```
1/1 [==============================] - 1s 1s/step
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/imagenet_class_index.json
35363/35363 [==============================] - 0s 1us/step
castle (34.03%)
```

Figure 1: Output 1

The above figure shows the Pre-Training of the model as a classifier.

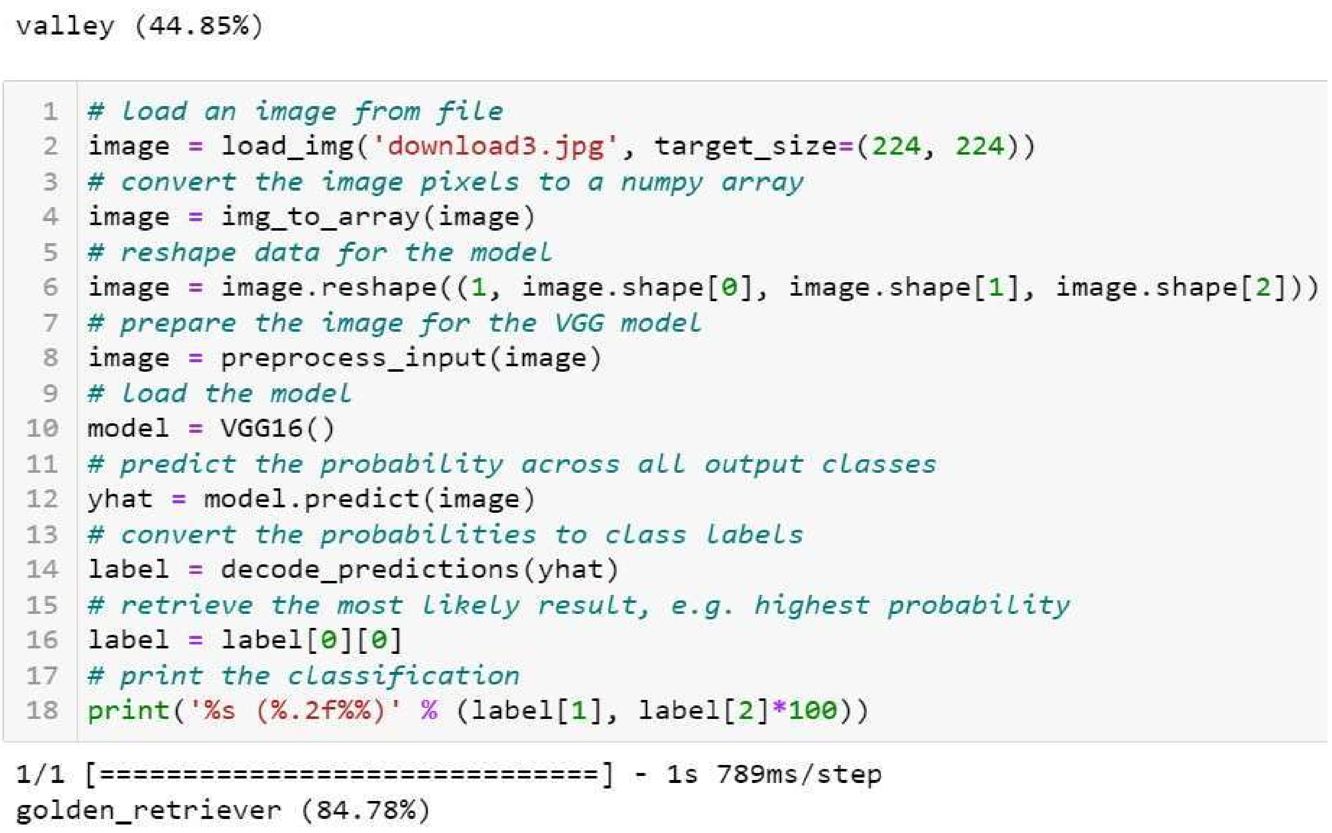


```
# load an image from file
image = load_img('download2.png', target_size=(224, 224))
# convert the image pixels to a numpy array
image = img_to_array(image)
# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)
# load the model
model = VGG16()
# predict the probability across all output classes
yhat = model.predict(image)
# convert the probabilities to class labels
label = decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0]
# print the classification
print('%s (%.2f%%)' % (label[1], label[2]*100))
```

```
1/1 [==============================] - 1s 806ms/step
valley (44.85%)
```

Figure 2: Predicition of 1st image

The above figure shows the prediction of 1st image which is valley(44.85%).



```
valley (44.85%)
```

```python
# load an image from file
image = load_img('download3.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array
image = img_to_array(image)
# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)
# load the model
model = VGG16()
# predict the probability across all output classes
yhat = model.predict(image)
# convert the probabilities to class labels
label = decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0]
# print the classification
print('%s (%.2f%%)' % (label[1], label[2]*100))
```

```
1/1 [==============================] - 1s 789ms/step
golden_retriever (84.78%)
```

Figure 2: Predicition of 2nd image

The above figure shows the prediction of 2nd image which is golden_retriever(84.78%).

**Conclusion:**

Thus, the model is been trained and implemented for Object detection using Transfer Learning of CNN. The model is been predicted and the results are:

1st image: valley(44.85%).

2nd image: golden_retriever(84.78%).